First Hit Fwd Refs
End of Result Set

Generate Collection Print

L23: Entry 1 of 1 File: USPT Nov 7, 2000

DOCUMENT-IDENTIFIER: US 6145007 A
TITLE: Interprocessor communication circuitry and methods

Abstract Text (1):
A method of exchanging messages between first and second processors. A pending flag in a first register is polled by the first processor and if the flag is in a first selected logic state, a message is written into a second register with the first processor. The pending flag is set to a second selected logic state with the first processor and an interrupt to the second processor is generated. The message is read from the second register with the second processor when the pending flag is in the second logic state. The pending flag set to the first logic state with the second processor.

Brief Summary Text (18):
According to the principles of the present invention, a method is provided for exchanging messages between first and second processors. A pending flag in a register is checked with the first processor. If the flag is in a first selected logic state, a message is written into a second register with the first processor. The first processor sets the pending flag to a second selected logic state and an interrupt to the second processor is generated. The message from the second register is read with the second processor when the pending flag is in the second logic state. The pending flag is set to the first logic state with the second processor.

Drawing Description Text (8):
FIG. 4 is a diagram of the interprocessor communications (IPC) registers as shown in FIG. 3.

Detailed Description Text (3):
FIG. 1A is a general overview of an audio information decoder 100 embodying the principles of the present invention. Decoder 100 is operable to receive data in any one of a number of formats, including compressed data in conforming to the AC-3 digital audio compression standard, (as defined by the United States Advanced Television System Committee) through a compressed data input port CDI. An independent digital audio data (DAI) port provides for the input of PCM, S/PDIF, or non-compressed digital audio data.

Detailed Description Text (15):
FIG. 1C is a high level functional block diagram of a multichannel audio decoder 100 embodying the principles of the present invention. Decoder 100 is divided into two major sections, a Processor Block 101 and the I/O Block 102. Processor Block 106 includes two digital signal processor (DSP) cores, DSP memory, and system reset control. I/O Block 102 includes interprocessor communication registers, peripheral I/O units with their necessary support logic, and interrupt controls. Blocks 101 and 102 communicate via interconnection with the I/O buses of the respective DSP cores. For instance, I/O Block 102 can generate interrupt requests and flag information for communication with Processor Block 101. All peripheral control and status registers are mapped to the DSP I/O buses for configuration by the DSPs.

Detailed Description Text (19):
IPC (Inter-processor Communication) registers 1302 support a control-messaging protocol for communication between processing cores 200 over a relatively low-bandwidth communication channel. High-bandwidth data can be passed between cores 200 via shared memory 204 in processor block 101.

Detailed Description Text (23):
In general, I/O registers are visible on both I/O buses, allowing access by either DSPA (200a)or DSPB (200b). Any read or write conflicts are resolved by treating DSPB as the master and ignoring DSPA.

Detailed Description Text (25):
The Host can choose between serial and parallel boot modes during the reset sequence. The Host interface mode and autobit mode status bits, available to DSPB 200b in the HOSTCTL register MODE field, control the boot mode selection. Since the host or an external host ROM always communicates through DSPB. DSPA 200a and 200b receives code from DSPB 200b in the same fashion, regardless of the host mode selected.

Detailed Description Text (27):
Usually, the software application will explicitly specify the desired output precision, dynamic range and distortion requirements. Apart from the intrinsic limitation of the compression algorithm itself, in an audio decompression task the inverse transform (reconstruction filter bank) is the stage which determines the precision of the output. Due to the finite-length of the registers in the DSP, each stage of processing (multiply+accumulate) will introduce noise due to elimination of the lesser significant bits. Adding features such as rounding and wider intermediate storage registers can alleviate the situation.

Detailed Description Text (30):
Debuggers can be of two kinds: static or dynamic. Static debugging involves halting the system and altering/viewing the states of the various sub-systems via their control/status registers. This offers a lot of valuable information especially if the system can automatically "freeze" on a breakpoint or other trapped event that the user can pre-specify. However, since the system has been altered from its run-time state, some of the debug actions/measurements could be irrelevant, e.g. timer/counter values.

Detailed Description Text (31):
Dynamic debugging allows one to do all the above while the system is actually running the application. For example, one can trace state variables over time just like a signal on an oscilloscope. This is very useful in analyzing real-time behavior. Alternatively, one could poll for a certain state in the system and then take suitable predetermined action.

Detailed Description Text (32):
Both types of debugging require special hardware with visibility to all the sub-systems of interest. For example, in a DSP-based system-on-a-chip such as decoder 100, the debug hardware would need access to all the sub-systems connected to the DSP core, and even visibility into the DSP core. Furthermore, dynamic debugging is more complex than its static counterpart since one has to consider problems of the debug hardware contending with the running sub-systems. Unlike a static debug session, one cannot hold off all the system hardware during a debug session since the system is active. Typically, this requires dual-port access to all the targeted sub-systems.

Detailed Description Text (33):
While the problems of dynamic debugging can be solved with complicated hardware there is a simpler solution which is just as effective while generating only

minimal additional processor overhead. Assuming that there is a single processor (like a DSP core 200a or 200b), in the system with access to all the control/state variables of interest, a simple interrupt-based debug communication interface can be built for this processor. The implementation could simply be an additional communication interface to the DSP core. For example, this interface could be 2-wire clock+data interface where a debugger can signal read/write requests with rising/falling edges on the data line while holding the clock line high, and debug port sends back an active low acknowledge on the same data line after the subsequent falling edge of the clock.

Detailed Description Text (34):
A debug session involves read/write messages sent from an external PC (debugger) to the processor via this simple debug interface. Assuming multiple-word messages in each debug session, the processor accumulates each word of the message by taking short interrupts from the main task and reading from the debug interface. Appropriate backup and restore of main task context are implemented to maintain transparency of the debug interrupt. Only when the processor 200a, 200b accumulates the entire message (end of message determined by a suitable protocol) is the message serviced. In case of a write message from the PC, the processor writes the specified control variable(s) with specified data.

Detailed Description Text (35):
In case of a read request from the PC, the processor compiles the requested information into a response message, writes the first of these words into the debug interface and simply returns to its main task. The PC then pulls out the response message words via the same mechanism--each read by the PC causes an interrupt to the processor which reloads the debug interface with the next response word till the whole response message is received by the PC.

Detailed Description Text (36):
Such a dynamic debugger can easily operate in static mode by implementing a special control message from the PC to the processor to slave itself to the debug task until instructed to resume the application.

Detailed Description Text (38):
Each processor in such a system will usually have dedicated resources (memory, internal registers etc.) and some shared resources (data input/output, inter-processor communication, etc.). A dedicated debug interface for each processor is also possible, but is avoided since it is more expensive, requires more connections, and increases the communication burden on the PC. Instead, the preferred method is using a shared debug interface through which the PC user can explicitly specify which processor is being targeted in the current debug session via appropriate syntax in the first word of the messaging protocol. On receiving this first word from the PC, the debug interface initiates communication only with the specified processor by sending it an initial interrupt. Once the targeted processor receives this interrupt it reads out the first word, and assumes control of the debug interface (by setting a control bit) and directs all subsequent interrupts to itself. This effectively holds off the other processor(s) for the duration of the current debug session. Once the targeted processor has received all the words in the debug message, it services the message. In case of a write message, it writes the specified control variable(s) with the specified data and then relinquishes control of the debug interface so that the PC can target any desired processor for the next debug session.

Detailed Description Text (39):
In case of a read request, the corresponding read response has to make its way back from the processor to the PC before the next debug session can be initiated. The targeted processor prepares the requested response message, places the first word in the debug interface and then returns to its main task. Once the PC pulls this word out, the processor receives an interrupt to place the next word. Only after

the complete response message has been pulled out does the processor relinquish the debug interface so that the PC can start the next debug session with any desired processor.

Detailed Description Text (42):
Whether static or dynamic, all the functions of a debugger can be viewed as reading state variables or setting control variables. However, traps and breakpoints are worthy of special discussion.

Detailed Description Text (44):
In the single-processor strategy, when the processor hits a trap it takes an interrupt from the main task, sends back an unsolicited message to the PC, and then dedicates itself to process further debug messages from the PC (switches to static mode). For example the PC could update the screen with all the system variables and await further user input. When the user issues a continue command, the PC first replaces the trap instruction with the backed-up (original) instruction and then allows the processor to revert to the main task (switches to dynamic mode).

Detailed Description Text (45):
In the multi-processor debug strategy, unsolicited messages from a processor to the PC are prohibited in order to resolve hardware contention problems. In such a case, the breakpoint strategy needs to be modified. Here, when a processor hits a trap instruction, it takes the interrupt from its main task, sets a predetermined state variable (for example, Breakpoint.sub.-- Flag), and then dedicates itself to process further debug messages from the PC (switches to static mode). Having setup this breakpoint in the first place, the PC should be regularly polling the Breakpoint.sub.-- Flag state variable on this processor--although at reasonable intervals so as not to waste processor bandwidth. As soon as it detects Breakpoint.sub.-- Flag to be set, the PC issues a debug message to clear this state variable to setup for the next breakpoint. Then, the PC proceeds just as in the single-processor case.

Detailed Description Text (49):
Decoder 100, as discussed above, includes shared memory of 544 words as well as communication "mailbox" (IPC block 1302) consisting of 10 I/O registers (5 for each direction of communication). FIG. 4 is a diagram representing the shared memory space and IPC registers (1302).

Detailed Description Text (50):
One set of communication registers looks like this

Detailed Description Text (51):
(a) AB.sub.-- command.sub.-- register (DSPA write/read, DSPB read only)

Detailed Description Text (52):
(b) AB.sub.-- parameter1.sub.-- register (DSPA write/read, DSPB read only)

Detailed Description Text (53):
(c) AB.sub.-- parameter2.sub.-- register (DSPA write/read, DSPB read only)

Detailed Description Text (56):
DSP B read only) where AB denotes the registers for communication from DSPA to DSPB. Similarly, the BA set of registers are used in the same manner, with simply DSPB being primarily the controlling processor.

Detailed Description Text (57):
Shared memory 204 is used as a high throughput channel, while communication registers serve as low bandwidth channel, as well as semaphore variables for protecting the shared resources.

Detailed Description Text (58):
Both DSPA and DSPA 200a, 200b can write to or read from shared memory 204. However, software management provides that the two DSPs never write to or read from shared memory in the same clock cycle. It is possible, however, that one DSP writes and the other reads from shared memory at the same time, given a two-phase clock in the DSP core. This way several virtual channels of communications could be created through shared memory. For example, one virtual channel is transfer of frequency domain coefficients of AC-3 stream and another virtual channel is transfer of PCM data independently of AC-3. While DSPA is putting the PCM data into shared memory, DSPB might be reading the AC-3 data at the same time. In this case both virtual channels have their own semaphore variables which reside in the AB.sub.-- shared.sub.-- memory.sub.-- semaphores registers and also different physical portions of shared memory are dedicated to the two data channels. AB.sub.-- command.sub.-- register is connected to the interrupt logic so that any write access to that register by DSPA results in an interrupt being generated on the DSP B, if enabled. In general, I/O registers are designed to be written by one DSP and read by another. The only exception is AB.sub.-- message.sub.-- semaphore register which can be written by both DSPs. Full symmetry in communication is provided even though for most applications the data flow is from DSPA to DSP B. However, messages usually flow in either direction, another set of 5 registers are provided as shown in FIG. 4 with BA prefix, for communication from DSPB to DSPA.

Detailed Description Text (59):
The AB.sub.-- message.sub.-- semaphore register is very important since it synchronizes the message communication. For example, if DSPA wants to send the message to DSPB, first it must check that the mailbox is empty, meaning that the previous message was taken, by reading a bit from this register which controls the access to the mailbox. If the bit is cleared, DSPA can proceed with writing the message and setting this bit to 1, indicating a new state, transmit mailbox full. The DSPB may either poll this bit or receive an interrupt (if enabled on the DSPB side), to find out that new message has arrived. Once it processes the new message, it clears the flag in the register, indicating to DSPA that its transmit mailbox has been emptied. If DSPA had another message to send before the mailbox was cleared it would have put in the transmit queue, whose depth depends on how much message traffic exists in the system. During this time DSPA would be reading the mailbox full flag. After DSPB has cleared the flag (set it to zero), DSPA can proceed with the next message, and after putting the message in the mailbox it will set the flag to I. Obviously, in this case both DSPs have to have both write and read access to the same physical register. However, they will never write at the same time, since DSPA is reading flag until it is zero and setting it to 1, while DSPB is reading the flag (if in polling mode) until it is 1 and writing a zero into it. These two processes a staggered in time through software discipline and.sub.-- management.

Detailed Description Text (60):
When it comes to shared memory a similar concept is adopted. Here the AB.sub.-- shared.sub.-- memory.sub.-- semaphore register is used. Once DSPA computes the transform coefficients but before it puts them into shared memory, it must check that the previous set of coefficients, for the previous channel has been taken by the DSPB. While DSPA is polling the semaphore bit which is in AB.sub.-- shared.sub.-- memory.sub.-- semaphore register it may receive a message from DSPB, via interrupt, that the coefficients are taken. In this case DSPA resets the semaphore bit in the register in its interrupt handler. This way DSPA has an exclusive write access to the AB.sub.-- shared.sub.-- memory.sub.-- semaphore register, while DSPB can only read from it. In case of AC-3, DSPB is polling for the availability of data in shared memory in its main loop, because the dynamics of the decode process is data driven. In other words there is no need to interrupt DSPB with the message that the data is ready, since at that point DSPB may not be able to take it anyway, since it is busy finishing the previous channel. Once DSPB is ready to take the next channel it will ask for it. Basically, data cannot be

pushed to DSPB, it must be pulled from the share memory by DSPB.

Detailed Description Text (61):
The exclusive write access to the AB.sub.-- shared.sub.-- memory.sub.-- semaphore register by DSPA is all that more important if there is another virtual channel (PCM data) implemented. In this case, DSPA might be putting the PCM data into share memory while DSPB is taking AC-3 data from it. So, if DSPB was to set the flag to zero, for the AC-3 channel, and DSPA was to set PCM flag to 1 there would be an access collision and system failure will result. For this reason, DSPB is simply sending message that it took the data from share memory and DSPA is setting share memory flags to zero in its interrupt handler. This way full synchronization is achieved and no access violations performed.

Detailed Description Text (69):
The Host interface mode bits and autoboot mode status bit are available to DSPB in the HOSTCTL register [23:20] (MODE field). Data always appears in the HOSTDATA register one byte at a time. The only difference in DSPB boot loader code for different modes, is the procedure of getting a byte from the HOSTDATA register. Once the byte is there, either from the serial or parallel interface or from an external memory in autoboot mode, the rest of DSPB boot loader code is identical for all modes. Upon determining the mode from the MODE bits, DSPB re-encodes the mode in the DBPST register in the following way: 0 is for autoboot, 1 for Ser. Mode, and 2 for Parallel Mode. This more efficient encoding of the mode is needed, since it is being invoked every time in the procedure Get.sub.-- Byte.sub.-- From.sub.-- Host. During application run-time, the code does not need to know what the Host interface mode is, since it is interrupt-driven and the incoming or outgoing byte is always in the HOSTDATA register. However, during the boot procedure, a polling strategy is adopted and for different modes different status bits are used. Specifically, HIN-BSY and HOUTRDY bits in the HOSTCTL register are used in the parallel mode, and IRDY and ORDY bits from SCPCN register are used in the serial mode.

Detailed Description Text (70):
Each DSP 200a, 200b has an independent reset bit in its own CONTROL register (CR) and can toggle its own reset bit after successful boot procedure. DSPA soft reset will reset only DSPA core and will not alter DSPA's MAPCTL, PMAP, and DMAP memory repair registers. DSPB soft reset will reset DSPB core as well as all I/O peripherals, but will not alter DSPB's MAPCTL, PMAP, and DMAP memory repair registers. Synchronized start is not an issue since the downloaded application code on each DSP handles synchronization.

Detailed Description Text (71):
Three major subroutines are described here. The first one is Get.sub.-- Byte.sub.-- From.sub.-- Host, which is mode-sensitive (checking is done on the encoded value in DBPTMP register). The byte is returned in the AR6 register.

Detailed Description Text (72):
The second subroutine is Send.sub.-- Byte.sub.-- To.sub.-- Host, which takes the byte in AR6 and sends it to the Host. This routine is not mode-sensitive, since when a byte is to be sent to the Host, the previous byte has already been picked up. This is true since messages returning to the Host are only byte-wide and only of two kinds, solicited or unsolicited.

Detailed Description Text (82):
The third important subroutine is Get.sub.-- Word.sub.-- From.sub.-- Host. This subroutine returns one 24 bit word in the COM.sub.-- BA register after using ACC0 and AR6 as a temporary storage. Actually, Get.sub.-- Byte.sub.-- From.sub.-- Host is invoked three times within Get.sub.-- Word.sub.-- From.sub.-- Host and the incoming byte in AR6 is shifted appropriately in ACC0. The Get.sub.-- Word.sub.-- From.sub.-- Host subroutine also updates the checksum by using ADD instead Of XOR.

The running checksum is kept in register PAR.sub.-- 2.sub.-- BA. Note that there is no Send.sub.-- Word.sub.-- To.sub.-- Host subroutine, since all replies to the Host are a full byte wide.

Detailed Description Text (83):
Interprocessor Communication (IPC) and Protocol can now be described in further detail in view of the discussion above and FIG. 4. The Dual DSP processor architecture according to the principles of the present invention, is advantageously very powerful in the effective use of available MIPS. However, it is important to remember that the target application must be such that it is relatively easy to split processing between the two engines. Both AC-3 and MPEG-2 multichannel surround applications possess this quality. The essential element to an efficient implementation of these applications is the effective communication between the two engines. In decoder 100 the shared resources between the two processors are the 544.times.24 word data memory 204 and the communication register file 1302 consisting of ten I/O registers.

Detailed Description Text (88):
The AB.sub.-- semaphore.sub.-- token register (FIG. 4) has the following format:

Detailed Description Text (89):
Note that DSPA can both write and read into this register and that DSPB can only read from this register.

Detailed Description Text (90):
The BA.sub.-- semaphore.sub.-- token register has the following format:

Detailed Description Text (91):
Note that DSPB can both write and read into this register and that DSPA can only read from this register

Detailed Description Text (92):
A. Communication Register File

Detailed Description Text (93):
The communication register file (FIG. 4) consists of eight registers. They are split into two groups of four registers each, as shown below.

Detailed Description Text (94):
The first group of four registers is used by DSPA to send commands to DSPB, along with appropriate parameters. The second set of registers is used by DSPB to send commands and parameters to DSPA. So, the communication protocol is completely symmetrical.

Detailed Description Text (95):
Consider the case when DSPA is sending a command to DSPB as shown in FIG. 5A. Before DSPA can send a command, it must check the COMMAND.sub.-- AB.sub.-- PENDING flag to make sure that the previous command from A to B was taken by DSPB (Step 5301). If it is appropriate to send the message (Step 5302), DSPA assembles the parameters, sets the COMMAND.sub.-- AB.sub.-- PENDING flag and writes the command itself (Step 5304). Otherwise, DSPA waits at Step 5303. The event of writing the COMMAND.sub.-- AB.sub.-- PENDING triggers a DSPB interrupt (Step 5305), which in turn reads the command and its parameters and at the end clears the COMMAND.sub.-- AB.sub.-- PENDING flag (Step 5306). This allows DSPA to then send another command if necessary.

Detailed Description Text (96):
FIG. 5B.sub.-- shows a substantially similar messaging process. In this case, DSPB polls the command pending but at Step 5308 to determine if a message is waiting, rather than receiving an active interrupt from DSPA.

Detailed Description Text (97):
It should be noted that both DSPs have write access to the COMMAND PENDING register but the software discipline will ensure that there is never a conflict in the access. If DSP(A/B) 200a, 200b cannot issue the command because the COMMAND.sub.--AB.sub.-- PENDING bit is set, it will either wait or put a message into a transmit queue. Once the command is received on the other side, the receiving DSP can either process the command (if it is a high-priority command) or store it into a receive queue and process the command later. Scheduling of command execution will be such that minimum latency is imposed in the system. Regular checking at the channel resolution (about 1 ms) will ensure minimal latency in processing commands.

Detailed Description Text (98):
When one processor is not accepting messages, a time-out is required to inform the Host about the potential problem. If DSPA is not responding to messages from DSPB, the Host will be notified by DSPB. If DSPB is not responding to DSPA, then, most likely, it is not responding to the Host either, and Host will know that explicitly. If DSPB is not responding to DSPA, but it is responding to the Host, DSPA will stall, will stop requesting data, the output buffers will underflow and the demux (or upstream delivery engine) will overflow in pushed systems or time-out in pulled systems.

Detailed Description Text (99):
The list of messages includes:

Detailed Description Text (100):
Messages from A to B:

Detailed Description Text (103):
3) bitstream status and info

Detailed Description Text (104):
4) decode status and info

Detailed Description Text (107):
Messages from B to A:

Detailed Description Text (109):
2) control and status messages that DSPA needs to execute on behalf of DSPB based upon the Host's request, including initialization and run time messages (setting up the registers in the input block etc.)

Detailed Description Text (111):
4) boot messages (in case that DSPB boots DSPA)

Detailed Description Paragraph Table (1):
TABLE 1
AB.sub.-- semaphore.sub.-- token register RD.sub.-- PRIVILEGE.sub.-- B WR.sub.-- USE.sub.-- A PCM.sub.-- DATA.sub.-- READY TC.sub.-- READY

Detailed Description Paragraph Table (2):
TABLE 2
BA.sub.-- semaphore.sub.-- token register WR.sub.-- PRIVILEGE.sub.-- A RD.sub.-- USE.sub.-- B BA.sub.-- DATA1.sub.-- READY BA.sub.-- DATA2.sub.-- READY

Detailed Description Paragraph Table (3):
COMMAND.sub.-- AB [23:0] PARAMETER.sub.-- 0.sub.-- AB [23:0] PARAMETER.sub.-- 1.sub.-- AB [23:0] COMMAND.sub.-- AB.sub.--

PENDING [0] AB.sub.-- semaphore token register COMMAND.sub.-- BA [23:0]
PARAMETER.sub.-- 0.sub.-- BA [23:0] PARAMETER.sub.-- 1.sub.-- BA [23:0]
COMMAND.sub.-- BA.sub.-- PENDING [0] BA.sub.-- semaphore.sub.-- token register

CLAIMS:

1. A method of exchanging messages between first and second processors forming a portion of a single-chip processing device, comprising the steps of:

checking a pending flag in a register with the first processor;

if the flag is in a first selected logic state indicating that a second register is ready for messaging, setting the pending flag to a second selected logic state with the first processor to indicate a message is being sent;

writing a message into the second register with the first processor;

generating an interrupt to the second processor;

reading the message from the second register with the second processor when the pending flag is in the second logic state; and

setting the pending flag to the first logic state with the second processor to indicate the second register is ready for messaging.

4. The method of claim 1 and further comprising the step of waiting with the first processor until the second processor sets the pending bit to the first logic state if after the step of checking the pending bit is in the second state.

5. The method of claim 2 and further comprising the step of polling the pending flag with the second processor to determine if the pending flag is in the second logic state.

6. The method of claim 1 and further comprising the step of generating an interrupt to the second processor when a message has been written into said second register.

7. A method of operating a single-chip audio decoder having first and second digital signal processors for decompressing compressed audio data and a set of interprocessor communications registers comprising the steps of:

checking a pending flag in a first one of the registers with the first processor;

if the pending flag indicates that a second one of the registers is available for exchanging messages, setting the pending flag with the first processor to indicate a message is being sent;

writing a message into the second register with the first processor;

reading the message from the second register with the second processor when the pending flag indicates that a message is being sent; and

resetting the pending flag in the first register with the second processor after said step of reading to indicate to the first processor that the second register is available for exchanging messages.

9. The method of claim 7 and further comprising the step of polling said first register with the second processor to determine if a message is being sent.

10. The method of claim 7 and further comprising the step of placing a second

message in a queue with the first processor when the pending flag indicates the second register is not available for messaging.

13. The method of claim 7 wherein the message comprises a status message.

14. The method of claim 7 wherein the message comprises a boot message.

15. An audio decoder fabricated on a single chip for decompressing compressed audio data comprising:

first and second communications registers;

a first processor operable to:

check a flag in said first register to determine if said second register is available for messaging;

changing said flag to indicate a message is being sent if the said flag indicates that said second register is available for exchanging messages; and

writing a message into said second register; and

a second processor operable to:

read said message from said second register with said second processor when said flag indicates that a message is being sent; and

reset said flag in said first register with the after reading said message to indicate to said first processor that said second register is available for exchanging messages.

17. The audio decoder of claim 15 wherein said first processor is further operable to send an interrupt to said second processor when a message is being sent through said second register.

18. The audio decoder of claim 15 wherein said second processor is further operable to poll said first register to determine whether a message is being sent through said second register.